Baa!

A procedural game based on real-time flocking behaviour

Bachelor's thesis DATX02-19-37

OSKAR GRÖNQVIST, ERIK MAGNUSSON,
IBRAHIM NABOULSI, LUCAS NORMAN,
MATILDA SJÖBLOM, MY SUNDQVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Bachelor's thesis DATX02-19-37

Baa!

A procedural game based on
real-time flocking behaviour

OSKAR GRÖNQVIST
ERIK MAGNUSSON
IBRAHIM NABOULSI
LUCAS NORMAN
MATILDA SJÖBLOM
MY SUNDQVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Baa! - A procedural game based on real-time flocking behaviour

OSKAR GRÖNQVIST,
ERIK MAGNUSSON,
IBRAHIM NABOULSI,
LUCAS NORMAN,
MATILDA SJÖBLOM,
MY SUNDQVIST,
May 2019.

Supervisor: Marco Fratarcangeli, Department of Computer Science and Engineering, Chalmers University of Technology

Examiner: Staffan Björk, Department of Computer Science and Engineering, University of Gothenburg

Bachelor Thesis DATX02-19-37
Department of computer science and engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2019

Abstract

Simulating the natural movement of large groups of flocking animals has been a growing field of research within computer science since the 1980's, largely because of its potential in performance optimisation. The purpose of this bachelor's thesis is to combine this existing research in flocking behaviour with gameplay design and optimisation to create a procedural game with flocking as its main feature. To accomplish this, Unreal Engine 4 and C++ has been used as a working environment. The result is an optimised game where roughly 700 sheep in a flock can be present without a significant performance loss. The actual number of sheep used in the game, however, is significantly lower to make the game more playable.

Sammanfattning

Att simulera rörelsemönster hos stora grupper av flockdjur har varit ett växande forskningsområde inom datavetenskap och datorgrafik sedan 1980-talet, mycket på grund av dess stora potential för prestandaoptimering. Syftet med det här kandidatarbetet är att kombinera den befintliga forskningen inom flockningsbeteende med speldesign och optimering för att skapa ett procedurellt spel med flockningsbeteende som utmärkande egenskap. För att åstadkomma detta har Unreal Engine 4 samt C++ använts som utvecklingsmiljö. Resultatet är ett optimerat spel där omkring 700 får i en flock kan simuleras utan signifikant påverkan på spelets prestanda. Det faktiska antalet får som används är dock betydligt lägre för att göra spelet mer spelbart.

Keywords: flocking behaviour, ai, video-game, unreal engine, sheep.

Acknowledgements

We would like to thank our supervisor Marco Fratarcangeli for his support and willingness to help us when we found ourselves lost in difficult tasks.

Group DATX02-19-37, Gothenburg, May 2019

Contents

List of Figures

List of Tables

Glossary

actor An object that can be placed in the game world according to Unreal Engine 4. xi, 18, 32

Autodesk Maya A 3D computer graphics application commonly used for interactive 3D applications, including video games, animated film, TV series, or visual effects. 14

boid A bird like object, coined by Craig Reynolds and is used to describe members of a flock. ix, 1, 2, 3, 9, 11, 13, 16

C++ A programming language based on C but with object oriented properties, commonly used in game development. 13, 32

mesh A collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modelling. 18, 32

pawn An actor that can be controlled by a player according to Unreal Engine 4. 18

pointer A variable that stores the address of a value in the memory so that it can be accessed directly in the memory. 16

thread A sequence of instructions for the CPU to execute successively. xi, 11, 16, 27, 32

thread pool A collection of threads that are assigned tasks to do in parallel.. 16

triggerbox A box shaped object in the game used to trigger events when overlapped with actors. 19

UE4 *Unreal Engine 4*. 3, 13, 14, 15, 27, 28, 32

1
Introduction

This project explores game development using procedural flocking behaviour through the creation of a sheep herding game based on existing theory on flocking behaviour algorithms, game design, and sheep behaviour. Various technical tools and methods are used and discussed, the main ones being Unreal Engine 4, spatial data structures, and parallelisation.

Humans have long studied the behaviour and movement patterns of large groups of animals and organisms. The complexity of hundreds, maybe thousands, of birds, sheep or even bacteria that move together in unison has been something that many researchers have aimed to describe. Although one might think this is solely a subject of study for biologists and other nature-oriented researchers, this phenomenon has caught the interest of researchers and professionals in computer graphics as well.

Due to the complexity of visualising large amounts of birds flying together in a flock, animators devoted their attention to research in flocking behaviour. Using a mathematical model to simulate flocking behaviour resulted in animations of flocking to both look more realistic and be easier to produce.

1.1 Work by Craig Reynolds

Few people have been so influential to a field of research as Craig Reynolds has been to flocking behaviour. After having practically invented the research field when publishing his 1987 article *Flocks, Herds, and Schools: A Distributed Behavioural Model* [1], his name has appeared in most articles related to flocking behaviour ever since. This report is no exception as most flocking in this game is built upon Reynolds' work that was published more than three decades ago.

1.2 Boids

Flocking behaviour does not only apply to birds, sheep, and bacteria. The simulation of flocking described in this report can be applied to many different kinds of animals, microorganisms and objects. To avoid naming a specific type of member when discussing flocking behaviour, Reynolds coined the term *Boid*, stemming from bird-like object, to refer to any type of member in a flock [1].

In the English language, there is a myriad of different words referring to groups of all types of animals. For that reason the remainder of this report is using the word *flock* to refer to any group of animals, including boids, just as Reynolds did in his article [1].

1.3 The flocking behaviour

Although a flock of birds can seem big, complex and incomprehensible it can, according to Reynolds, be simulated using three simple rules [1].

- Collision Avoidance: the separation of the flock members to avoid collision.
- Velocity Matching: attempt to match velocity with nearby flockmates.
- Flock Centring: attempt to stay close to nearby flockmates.

These are the rules that each individual boid in a flock needs to follow in order to collectively achieve a flock-like behaviour. Each rule, along with its usefulness and difficulties, is described in more detail in section 2.3.

1.4 Practical Applications

A simulated flocking behaviour can be applied in many scientific and non-scientific areas. The animators of the movie *Batman Returns*, which premiered in 1992 [2], wanted to simulate a colony of black bats that swarmed as a group down the streets of Gotham City. To achieve this behaviour, they created one bat that looked as they wanted it to and then copied this bat until they had a colony that was big enough. After this, they made all the individual bats have an average velocity of nearby bats, go in the average direction and not get too far away from their neighbours. The result was a relatively realistic colony of bats.

Flocking has been used in other areas besides animating a flock of animals for films and other media. A more serious usage is in the military, where a swarm of autonomous drones can be instructed to attack a group of people [2]. Another military area the US army has considered using flocking for is its military satellites. The idea is that if a satellite is divided into several small mobile units, it is harder to target and take down by foreign opponents [2].

Other areas where flocking algorithms have been tested include pathfinding for real-time strategy games, such as StarCraft. One study applied flocking for micromanagement in StarCraft, where micromanagement means placing small units at strategic places in the game world [3]. Unfortunately however, it was not considered significantly better than what had previously been used and was deemed too ineffective to be used [3]. Another study applied Reynolds' flocking behaviour on bacteria and interestingly found that, even if bacteria lack a central nervous system

as for example birds, the simulation behaved almost like real bacteria when the three flocking rules were applied [4].

1.5 Purpose

The ambition for the project is to develop an entertaining procedural game with flocking behaviour as the central feature. During the development, it is also investigated whether a flocking algorithm can be the central feature of a game designed using existing gameplay design theory.

The objective is to design a game based on flocking behaviour where the main focus is its gameplay and player experience while maintaining a behaviour that resembles flocking in nature as closely as possible. Different ways of optimising the flocking algorithm is explored in order to achieve a group of up to 1000 boids being run simultaneously without experiencing any severe performance issues.

1.6 Scope

As a requirement of the project is procedurality, it was decided that the randomness of flocking behaviour would fulfil this condition but that other applications of procedurality such as level generation is not included in the scope. The reason for this is that the focus mainly lies in making a well developed flocking behaviour that behaves as intended. Another reason is the limitation of time and resources. Making the flock procedural increases its realism making it easier to see if the individual actors perform as they should.

The visual part is created with *Unreal Engine 4* (UE4) as this game engine gives a lot of assistance when constructing the graphics. Using a graphics library such as *OpenGL* would allow for greater optimisation but the extra effort is not constructive towards the goals of the project. The result should be a demo of a game concept, as developing a full game in the given time-span does not seem plausible. The optimisation for large flocks is secondary to the gameplay, both because making a fun and realistic game is more important to our purpose, and making the flocks too large would affect the playability of the game.

2
Theory

This chapter aims to give the reader an overview of the theory behind simulating flocking behaviours, as well as an introduction to concepts and tools commonly used in game development. In addition to this the chapter introduces sheep specific behaviours which should be taken into consideration when adjusting the simulated flocking behaviour specifically for a herd of sheep.

2.1 Game Engines

A game engine works as an abstracting layer between the computer and the game creator. By providing facilitating features, it makes the process of developing the game easier. These are the features that game engines usually facilitate [5]:

- Render graphics. The first impression a player gets from a game is how it looks. With modern game engines, rendering powerful graphics is one of its many benefits. One can import assets from different platforms as the game engine makes the importing fast and accurate.
- Physics. To have realistic physical objects is important in order to create a good game experience for a player. A game engine helps with creating objects that have physical restrictions and are able to move in a world without having to code every step.
- GUI. Graphical user interfaces are sometimes crucial to make the game more understandable for a player. Game engines support making a graphical user interface with different kinds of editors designed for that specific purpose.
- Scripting. Many game engines have pre-made scripts that for example handles camera movement.
- Networking. If a game is going to have multiple players over a network, it will need server power. Some game engines provide workflows to make it easier to have stable network functionality.
- Sounds. Most games are going to have some sound effects, these are easy to integrate and assign to specific events by using a game engine.

2.2 Game Design

Game design is about the creators' process of decision making when constructing a game. In order to get the user to understand how the game should be played and which rules that make the game, the designer needs to ensure that the game shows clearly the way to play it [6].

Initial research into the area of game design highlighted the importance of balancing the game experience. In the book *The Art of Game Design: a book of lenses* by Jesse Schell, he compares creating a balanced game with creating a recipe [6, pp. 175-205]. When creating a recipe, all the ingredients are decided, but not how much of each in order to get an appetising meal. The same process is important when balancing games, all the elements may be there but their appropriate ratios have to be determined in order to get a fun and entertaining game. In this section, the game patterns and elements that this game consists of is described.

2.2.1 The Game World

The game world is where the game takes place. It describes the spatial relationship between the game elements, such as restricting where the player can and cannot move. There are two kinds of game worlds, discrete and continuous. Continuous is when the player appears to move seamlessly without obvious restrictions, such as in open world games. Discrete is when the player moves in distinctly defined levels, where the player has to complete a level to access further areas of the game world. A game world can be constructed in many ways; 2D, 3D, linear, non-linear, et cetera. For this game, a 3D approach has been taken, meaning that the player and the game elements are placed and can be moved in three dimensions.

Another part of the game world is levels, in which player needs to fulfil certain criteria for or in order to get to the next one. Levels can use certain *closure points*, which is forcing the player to do irreversible actions [7, pp. 55-107]. By crossing closure points, the player cannot undo their choice, thus making the action irreversible.

A game world can also take usage of inaccessible areas. Inaccessible areas are areas that the player can not currently enter. Even if the player is not able to currently access the area, they may be able to affect and view it [7, pp. 55-107]. Inaccessible areas are commonly used in cases where the level or world is of a limited size, and the creators want it to appear bigger. An area of this kind is often blocked by something that makes the player unable to access it, this can be another game element like a high fence, stones, doors, and so forth. However, sometimes other actors in the game can reach the area; for example, if there is a river blocking the way and the player is unable to swim but other actors are. In some cases, what is blocking the area is not visible. For example, if a certain goal has to be reached in order to be able to move through.

2.2.2 Score

A score is an element in games that motivates the player to investigate what can be done in the game by changing it, and by that, explore the game itself more. The game designer needs to define what will generate a higher or lower score, and by how much, as the score is somewhat representing success in the game. If the player gets enough information, they can calculate her own tactics in order to get a certain amount of points in the fastest way or get an as high or low score as possible [6], [7].

2.2.3 Resources

Resources in games are used to represent some kind of commodity the player has to manage [7, pp. 107-122]. It has to be clear to the user what the resources are there for and which actions the player can perform by utilising them. The need to manage the resources can encourage the player to think strategically, hence it is a common element in games to make the resources both limited and renewable.

2.2.4 End Condition

In many arcade games, the level of success is measured by how long the player manages to survive. In other words, the game will always end based on some condition [7]. In order to add value to the game experience, an end condition needs to be set that gets increasingly tougher for the player to handle. This creates tension between the player and the game. If the player discovers that there is a way of losing, the tension rises, the challenge increases and value to the game is created [6]. This is a way of balancing a game; if it is too easy to play a game forever without losing, there is no challenge. On the other hand, if the player keeps losing at an early point in a game, it can be frustrating to play.

2.3 Flocking Simulation

A flocking behaviour emerges when many separate agents group together to form a larger entity despite all decisions being made individually. In computers, such a distributed model allows for easier parallelisation and does not require a higher performance computer as a more centralised model would. The following section will describe Reynolds' three rules [1] as well as a few other rules which create a more realistic behaviour, such as collision avoidance and visual fields.

2.3.1 Reynolds' Three Flocking Rules

As previously mentioned in section 1.1, the research field of flocking behaviour simulations was largely created when Craig Reynolds published his 1987 article [1], making his work and accomplishments popular to recreate. This includes Reynolds' three rules for flocking behaviour, collision avoidance (from now on referred to as separation of members), velocity matching and flock centring [1]. What Reynolds discovered was that while flocking in real animals is a very complex process that

has been developed over millions of years, it can be simulated by calculating and summing-up acceleration vectors for each of the three rules. The following descriptions of the rules are largely based on Reynolds' article.

Separation is a boid's desire to stay at a safe distance away from its fellow flock mates to avoid colliding with each other [1]. This is often implemented by applying a force opposite the direction of another boid that is inversely proportional to the distance between the boids. In other words, the closer a boid is to another boid, the stronger is its desire to move away from that boid.

Velocity matching, or alignment is the desire to move in the direction of the rest of the flock [1]. This is often implemented by finding the average of the other boids' velocity vectors within its visual field. This rule makes it possible for smaller flocks or individual boids to merge with larger flocks.

Flock centring, or cohesion, is a boid's desire to stay close to its flock mates and form an actual flock [1]. While this may sound contradictory to the first rule, using only one of them would result in a desire to *either* get closer to its flock mates *or* move away from them. A combination of these two rules create a desire to stay within a certain distance or distance interval from its flock mates. This rule is often implemented by creating an acceleration vector in the direction of the nearby boids which is proportional to the distance between them.

2.3.2 Modifications in Previous Work

Ever since Reynolds' original flocking algorithm from the 1980s there have been many modifications both to the original three rules and by introducing additional rules. One such modified flocking algorithm is the one introduced in *Extending Reynolds' flocking model to a simulation of sheep in the presence of a predator*, which presents an algorithm modified to work with sheep and a predator [8].

The modified algorithm has minor changes to the three original rules. Reynold's cohesion implementation performs calculations on a limited number of boids [1], while the modified algorithm performs the calculation on all boids or sheep in order to ensure that the sheep come to a stop. The result of the separation rule is normalised in order to get just the direction and not the strength. The only change made to the alignment rule is that it is given very small importance when weighted in order to avoid constant motion [8].

When the sheep enter the flight zone, which is a radius around the predator, the escape rule based on formula 2.1 is activated. In order to avoid the activation being too sudden, it uses an inverse square function.

$$esc(s) = \frac{s_p - p_p}{|s_p - p_p|} inv(|s_p - p_p|, 10) \text{ [8]} \tag{2.1}$$

Where:
- s_p: is the position of the sheep
- p_p: is the position of the predator
- $inv(x, s) = (\frac{x}{s} + \epsilon)^{-2}$
- ϵ: a small value in order to avoid division by 0

In addition to these modifications the weights of the original rules are different whenever a predator is close to a sheep, as can be seen in formula 2.2. The weights are shifted using a sigmoid function $p(x)$ where x is the the distance to the predator in order to get a smooth transition between the different weights.

$$\begin{aligned} v = & m_c(1 + p(x)m_{cp})coh(s) + \\ & + m_s(1 + p(x)m_{sp})sep(s) + \\ & + m_a(1 + p(x)m_{ap})ali(s) + \\ & + m_e esc(s) \text{[8]} \end{aligned} \tag{2.2}$$

Where:
- m_f: is a weight for the force f
- m_{fp}: is a weight for the force f used when a predator is near

After the formula 2.2 the velocity is capped to a maximum velocity which increases when the predator comes closer.

2.3.3 Collision Avoidance

An important aspect of achieving realistic flocking behaviour is the boids' ability to detect and interact with their environment. Collision avoidance is an example of that. Collision avoidance is not to be confused with collision detection, however, as the latter is merely what triggers the algorithm of the former. When the boid is approaching an object which it is going to collide with, the boid's behaviour needs to change depending on its distance to that object.

The algorithm's first response should be to give it a desire to move away from the object by applying an acceleration inversely proportional to the distance parallel to the impact normal (the vector perpendicular to the point of future impact). Since there are many forces from many different sheep in play however, it is necessary to eliminate all movement towards the object as some point making a collision impossible.

2.3.4 Perception and Visual Fields

An aspect touched upon by Reynolds in his 1987 article is the use of different kinds and shapes of visual fields to change a boid's perception [1]. The most simple implementation of this would be making a boid aware of everything within a certain

radius of itself, including other boids and objects. This approach can, along with well tuned flocking rules, produce a satisfying result but is not often the most realistic representation of an animal's vision. When taking inspiration from nature, one can see that many prey animals have their eyes placed on the sides of their heads meaning they have a wider visual range, sometimes as wide as 360 degrees, while predators often have eyes placed on the front to improve depth perception while decreasing their field of view [9]. What makes more complex visual fields harder to implement is largely a problem with performance. Since a circular visual field is the easiest and fastest possible implementation, any deviation from that shape creates additional calculations and performance drops.

2.3.5 Sheep Behaviour

As this project aims to create a game based on herding sheep, tuning of the flocking behaviour has been made. Sheep are different from the standard boid in that the simulation of their behaviour does not require three dimensions. They also have a very wide range of vision, the only blind spot they have is right behind them [10]. Their binocular vision; the vision using two eyes to focus and see depth, is narrow. On the other hand, their peripheral vision is very wide as their eyes are on the sides of their head.

When sheep are herded, studies show that they tend to move closer to the centre of the flock compared to when they flock without external influence [11]. The closer a predator appears to be, the closer the sheep seek the centre. This study also meant that if the flock was very scattered, they all tended to seek the middle, creating a more tight cluster of sheep. Besides these findings, it appears that if a sheep leaves the flock when the group is standing still, the other members of the flock are going to follow the departing sheep [12]. Researchers also observed that if many sheep have left the flock, the remaining sheep in that flock will be more inclined to separate from the group and depart at an increased frequency.

2.4 Performance Optimisation

Simulating a large herd of sheep is an expensive task for a computer to handle. When calculating the next position for each sheep, the separation, cohesion, and alignment forces must be calculated for each sheep. Furthermore, each sheep needs to look for its adjacent sheep to calculate these forces. This would mean going through a list of all sheep and comparing the distance between them with the sheep's field of view to determine whether the adjacent sheep are to be considered when calculating the forces. These are expensive calculations for a single sheep in the herd and would yield performance issues when expanding the herd.

Many different techniques can be used to optimise the flocking algorithm for the herd of sheep. This section will describe two critical methods. The first is optimising with spatial data structures in order to make access to adjacent sheep cheaper.

Second is introducing parallelism in order to perform the calculations for Reynolds' three rules in parallel for all sheep in the herd.

2.4.1 Optimising with Spatial Data Structures

The high complexity of the algorithm executing the flocking behaviour can be largely decreased using a spatial data structure. Spatial data structures subdivides the space into pieces where only information about the boids that are currently inside that subdivision is saved.

Since the boids are spatial data and a lot of operations want to find boids close to another boid, an algorithm that does not use spatial data structures will have a quadratic complexity ($O(N^2)$) [13]. This method is a so-called brute force algorithm, that compares every pair of boids in nested for loops. By using a spatial data structure, this operation will be much faster. A spatial data structure, such as *Uniform Grid* or *k-d trees* (k-dimensional trees), partitions the space into smaller sections [14]. This makes it so that each boid only has to consider other boids that are located in subdivisions close to it. With an efficient spatial data structure it is possible to reach close to a constant complexity.

2.4.2 Optimising with Parallelisation

Collision detection is usually a slow operation. As every boid in the flock has to be compared with every other boid, it has a complexity of O(N^2) when doing it in a single thread [15]. Moreover, it is not just the collision detection that has to be calculated for every boid, it is all the other rules that the flocking behaviour consists of as well. Since these calculations have to be done for every boid and the calculations are independent from each other, the problem is very suitable for parallelisation. When parallelising, the problem is subdivided into separate threads of the CPU, central processing unit, or the GPU, graphics processing unit. This means that an amount of boids are put on separate threads. By this, the calculation of each boid can be done asynchronously, meaning that the computer does not wait until a calculation of boid n is done before starting with the next boid [16]. A purely computational problem such as flocking can be called an embarrassingly parallel problem, meaning it can be parallelised relatively easily. There could, however, potentially be problems with other parts of the program, such as the game engine which may result in the parallelisation not being as effective as it theoretically could be.

3
Methods

This chapter presents all the tools and methodology used for creating this project. This includes everything from how the division of the project has been made, to which algorithms and tools that were used during the implementation.

3.1 Tools

To render all of the boids on screen, a graphics library or a game engine is needed. There are plenty of options to choose from like the graphics libraries OpenGL and libGDX, or the game engines Unity and Unreal Engine 4. In order to put more focus on gameplay rather than rendering, a game engine was used. As the group was interested in learning to program in C++, UE4 was chosen over other game engines.

Unreal Engine 4 is a game engine developed by Epic Games. It is free to use for everyone unless one uses it for commercial use, then five per cent of the income has to be paid to Epic Games [17]. UE4 provides rendering, physics and garbage collection for the developers [18]. Besides writing pure C++ code, UE4 gives the user the option to utilise visual scripting through Blueprints. In this project, Blueprints are used mainly for user interfaces and variables that need to be set in the editor.

3.2 Creating a Simulation

Even though the final task is to create a game, a simulation was first made as a proof of concept. When making the calculations for each of the three rules that the flocking behaviour mainly consists of, the results will be three different, often conflicting, behaviours. One of the problems is to balance the forces of these conflicting behaviours to simulate a realistic flock. For example, if a boid's desire to stay with the flock is correctly balanced with its desire to stay away from nearby individuals such that the boids of the flock maintain a reasonable distance between each other.

The three rules defined by Reynolds is a solid base for a realistic looking flock but there is room for improvement. Other implementations have made changes by for example introducing additional rules in order to make the flock mimic a certain unique behaviour. The first implementation of flocking can be seen in figure 3.1, where a set of boids are flying in the air together and creating a flocking behaviour.

Figure 3.1: This project's first simulation of a flock of boids.

3.3 Creating the Game

When the simulation was proven to work the next task was to make a game where flocking plays a large role. This comes with a few problems that are not directly linked with flocking, such as designing a fun and interesting game and making it run alongside the flocking behaviour. In order to achieve this, certain tools were used to help with designing, optimising and building.

3.3.1 Graphic design

The graphical components of the game follow a consistently stylised design, chosen for practical reasons as well as its aesthetic appeal. The game features relatively low polygon models, particle systems for ambient visual effects, vector graphics and an easy to read sans serif font.

The character models chosen for the game have a relatively low poly count at ~280 triangles. The pre-animated models were purchased from the Unity store as part of a farm animal pack [19] containing sheep, goats, chickens and other animals that would be found at a farm. The pack did ,however, not include a dog model, so one had to be created by making adjustments to one of the existing animal models using *Autodesk Maya*. Environment models such as the fences [20], trees and rocks [21] were downloaded as part of free model packs sourced from the indie game marketplace itch.io and Blend Swap. Some adjustments were made to reduce the number of polygons and re-export the models with their positions at origin to make it easier to place the models in the UE4 level editor.

The intention of rendering many actors at the same time made low poly models an easy choice. Not only is it an appealing visual style but it is practical from a

performance perspective as it will require less GPU processing power to render the individual actors. The models used in the project have a sufficiently low poly count for the purpose of game. If the game was to be optimised to allow actors in the numbers of several tens of thousands, models with significantly lower poly counts would have been needed. It is, however, likely that this would have prompted a change of the viewing angle to an orthogonal top down view rendering 2D sprites a better option for the representation of the sheep.

(a) Without cel shading **(b)** With cel shading

Figure 3.2: A scene comparison between a) default materials and b) with cel shading. The cel shading gives the scene a cartoon look with blocks of colour.

Cel shaders, also known as toon shaders, mimic the visual style of cartoons. This effect is achieved by computing lighting per pixel and quantizing to a discrete number of colours which has the effect that the objects shadows and highlights appear as blocks of colours [22, Ch. 15.1], as seen in figure 3.2. The game implements cel shading through a UE4 post processing material. The material contains a light buffer which contains information about how lit a surface is based on colour values collected from the post process input and diffuse colour buffers. These values are then clamped to a range between zero and one, and compared to a look up table in the shape of a greyscale gradient texture that determines the brightness of the diffuse colour.

Many implementations of cel shaders include a black outline which helps to enhance the cartoon look [22, Ch. 15.1]. To maintain the softer impression achieved by cel shading without outlines, they were not included in the final implementation.

While the aim was to achieve a stylised look with the cel shader, the flat shading made it more difficult to visually separate the sheep in the flock. In order to

better represent the 3D objects, and make them pop in the world, dynamic ambient occlusion was activated in the details of the UE4 camera actors.

Dynamic ambient occlusion is a real-time approximation of how occluded the points of a surface is from ambient light [22, Ch. 11.3.5]. The more occluded a surface area is, the darker the lighting will be, resulting in softer, more realistic looking shadows. The dynamic ambient occlusion offered as a feature in Unreal Engine 4 is an implementation of screen-space ambient occlusion [23]. Screen-space ambient occlusion can be implemented in a few different ways, but in their simplest form it performs an effective approximation of the occlusion per pixel by repeatedly sampling the depth buffer for nearby pixels to form a simple representation of a models occlusion [22, Ch. 11.3.6].

The intensity of the ambient occlusion in the game level was set to a high level to achieve dark but soft shadows which helps the models to stand out against the rest of the scene. This intensity level would likely be exaggerated in another setting, but with out stylised visuals it serves well as a tool to increase the visual readability.

3.3.2 Parallelisation

There are two ways of implementing thread management in Unreal Engine 4, by using *FRunnable* and *FAsyncTask* which are built in thread management classes in UE4. The optimal choice for our simulation is to use FAsyncTask since these types of tasks are non blocking and will not interrupt the main game thread which would otherwise stop our game logic and cause performance issues. Additionally, assigning a task to a thread pool can be done faster with FAsyncTask compared to FRunnable which has more delays and overhead.

In order to perform the flocking algorithm for each boid, an array of pointers to all boids is sent to a class which purpose is to assign threads in a thread pool with tasks to perform the flocking algorithm for every boid.

3.3.3 Uniform Grid

Searching for sheep that are close to another sheep is an expensive task as it requires sifting through an array of all sheep that are present on the map. This is an important task as it is required when performing collision avoidance and flocking rules calculations for each individual sheep. The task can be completed more efficiently by using a uniform grid where 2D space can be divided into cells of equal width and height where each cell may contain one or more sheep from the herd (See figure 3.3) [14]. This means that e.g sheep *s4* only has to look for other sheep that are in its own cell and in adjacent cells (*s1,s2,s3*). The sheep that are further away from *s1* (*s5,s6*) can be ignored as they will not have an impact on the calculated forces that will be acting on *s1*. It is also safe to ignore them when considering collision avoidance as they are too far away from the sheep to collide with them.

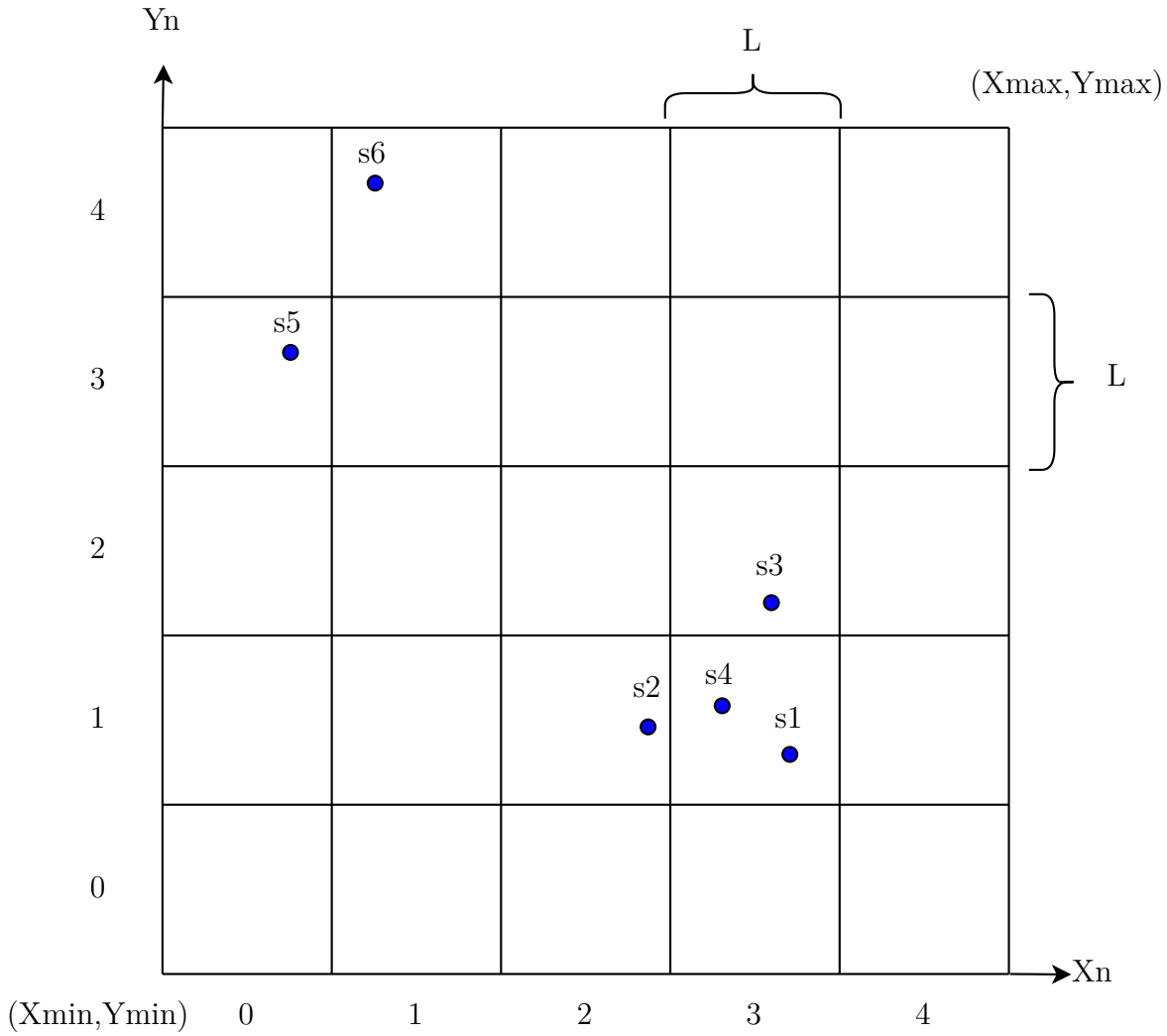


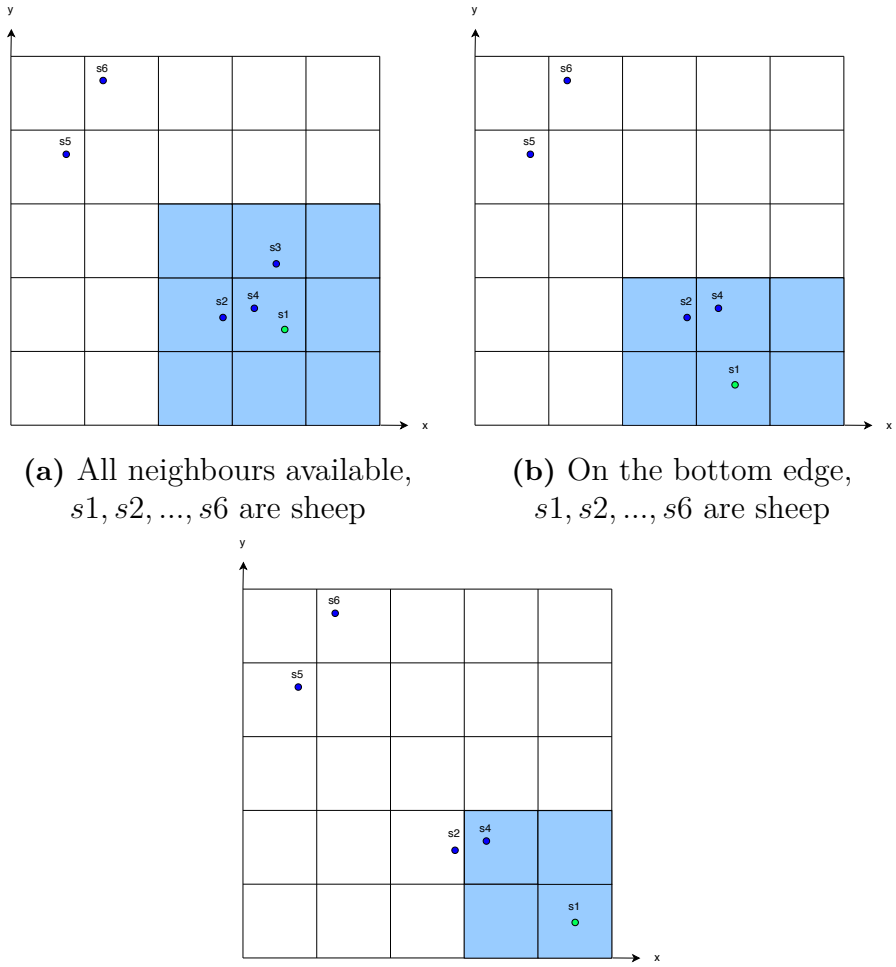
Figure 3.3: A 5x5 uniform grid where $s1, s2, ..., s6$ are sheep

The grid will be located in an arbitrary position on the map, therefore calculating which cell a sheep belongs to can be done in the following way:

$$x_n = \left\lfloor \frac{x_s - x_{min}}{L} \right\rfloor, y_n = \left\lfloor \frac{y_s - y_{min}}{L} \right\rfloor$$

Where (x_{min}, y_{min}) is the grid's coordinates, (x_s, y_s) are the sheep's coordinates relative to the map's coordinate system in game and L is the width and length of each cell. (x_n, y_n) are the cell's coordinates relative to the grid itself as seen on figure 3.3.

The reason the adjacent cells need to be checked is because one sheep could be positioned on the edge of its own cell and its direction is heading towards another cell. If the adjacent cells are not considered in this case, the sampling pool would be too small for the flocking behaviour and collision avoidance algorithms to work properly. The adjacent cells that are checked for each sheep are highlighted in figure 3.4 where sheep *s1* is the point of reference.



(a) All neighbours available, $s1, s2, ..., s6$ are sheep

(b) On the bottom edge, $s1, s2, ..., s6$ are sheep

(c) On the bottom right edge, $s1, s2, ..., s6$ are sheep

Figure 3.4: Highlighted nearby cells for a sheep

3.4 Game Design

Objects and non-controllable characters that can be placed in the game world are called Actors within Unreal Engine 4. In this game, all sheep and all stationary 3D models, like the fence and trees, are actors [24]. Objects that can be controlled by a user are called Pawns, and in this game, the dog is a pawn [25].

The game world is implemented by a landscape object as a base, that can be placed in a 3D world using Unreal Engine's editor. Every level in the world is distinguished by placed actors with fence meshes as paddocks and fixed cameras to get a satisfying overview of the level-areas, this can be seen in figure 3.5 where all the yellow dots are cameras. To make it clear to the player when the level is changing, the camera view changes to one that belongs to the level that the player is currently entering. By distinguishing the levels like this, the game world becomes discrete.

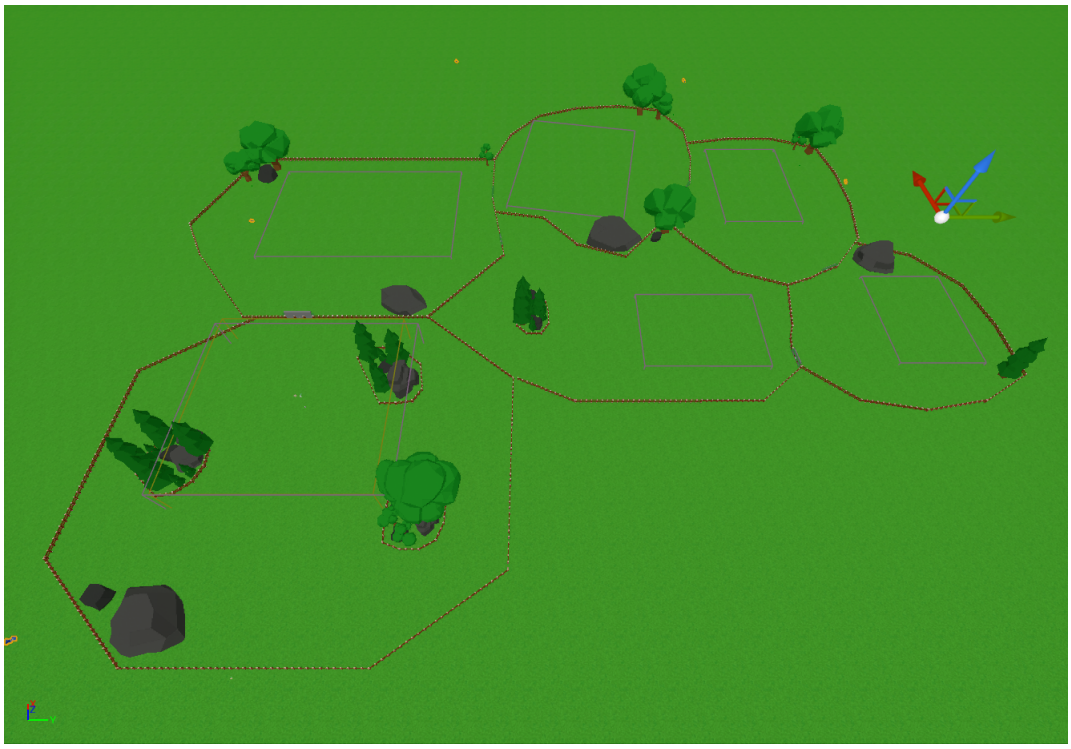


Figure 3.5: The whole game world from a top down view.

The score is what defines the player's success in the game. When the player herds sheep into a new level, they initially gets ten points for each sheep. To encourage the player to play for as long as possible and thus making it to an as high level as possible, the score that the player gets is multiplied by the current level number.

The initialisation of the first level starts a timer which remains active for the duration of the game. If the player has not advanced to the next level before the time has run out, the game ends. On the other hand, if the player manages to the next level in time, the time left increases in order to reward the player and make it possible to play the game for a longer time.

3.4.1 Gate Functionality

The gates in the game manage many different gameplay elements, they make sure neither the flock nor the dog can move to levels they are not supposed to, switch camera, and count sheep crossings. The gates function by three different parts, two triggerboxes per gate for detecting crossings as well as crossing direction, an animated gate to block access for sheep and the dog, and an invisible wall to block premature access for the dog. Two triggerboxes are used in order to tell which direction the crossing actor is heading, upon contact with one of the triggerboxes the game tries to add the actor to a list belonging to the gate. When an actor is already in the list, the game knows that the actor has previously made contact with

one of the triggerboxes in the gate pair, meaning the actor is leaving that area and is now entering the area belonging to the now activated triggerbox. The invisible wall which hinders the dog from entering the next level is necessary because the dog should not be able to transition before enough sheep have been herded, when the requirement is fulfilled the invisible wall is deleted. This is the game's inaccessible areas, as described in the section 2.2.1. They are used in order to make the world appear bigger as well as showing the player where to go next.

3.4.2 Balancing the Game

As described in section 2.2 it is important to have a game that is well balanced. The concept of resources in games are described in section 2.2.3 and such game elements can be used to balance games. In this game, the resources are the stamina which enables the dog to sprint for a short time, the deployable fence-barricades that can be placed in the world by the player , and the timer. All of these resources are limited and renewable; the stamina for the dog decreases and when it is not used it is regenerated and the barricades can only be placed five at a time. The timer is started in the beginning of the game and is increased when the player advances to the next level.

The barricades can be placed in the world by the player in order to control the sheep remotely which allows for crowd control, hence helping with herding as many sheep as possible through the gates. The dog-sprint resource helps the player to manoeuvre the dog around the herd of sheep as the dog is moving faster when sprinting. When herding a higher amount of sheep it is advantageous to use the barricade resource combined with the dog sprint. Thus, the player can make up different strategies in order to achieve as a high score as possible before the time run out. If these resources were not limited, it would have been too straightforward to herd a high amount of sheep. This would have made the game unbalanced, and therefore these resources are limited but still renewable. On the other hand, if the resources were not renewable, the game could have been too difficult and thus making it unbalanced.

3.5 Flocking Implementation

The flocking behaviour in the game is based on Reynolds' three rules as well as some modifications from Barksten and Rydberg's bachelor essay [8] as described in section 2.3.2, which introduces different behaviour for sheep close to predators such as, but not limited to, escaping.

The flocking implementation introduces a set of tuneable variables that alter the flocks behaviour. In order to achieve a satisfactory flocking behaviour these variables had to be adjusted manually, to make this process simpler they were made editable in the Unreal Engine 4 editor as well as in runtime, see table 3.1 for all the parameters.

Table 3.1: Tuneable parameters for the flocking behaviour

Parameter name	Description
maxForce	Maximum limit on the forces acting on the sheep
minSpeed	Minimum speed of the sheep
maxSpeed	Maximum speed of the sheep
searchRadius	Radius of the sheep's field of view
mc	Weight of the collision rule
mcp	Weight of the collision rule in presence of dog
ms	Weight of the separation rule
msp	Weight of the separation rule in presence of dog
ma	Weight of the alignment rule
map	Weight of the alignment rule in presence of dog
me	Weight of the escape rule
mAvoid	Weight of the avoid rule
escapeRadius	Radius around dog where sheep are affected by it
escSoft	Softness of escape rule
sepSoft	Softness of separation rule

3.5.1 Perception and Visual Fields

Creating a more realistic visual field was given a low priority since there was no time to implement it and as declared in section 2.3.5 sheep have close to a complete 360 degree field of vision. This means that the game is using a circular, or spherical visual field where the distance between the boids is compared with a certain radius. The method used for this is called *SphereOverlapActor* and uses overlapping between an invisible sphere around a boid and another boid to determine whether the latter is within the visual field of the former [26].

3.5.2 Collision Avoidance

Predicting potential collisions for the boids is achieved by line tracing. This is done by giving all the boids invisible vectors in front of them and on the sides (see figure 3.6). If the vector overlaps any other object except the neighbouring boids, a collision avoidance algorithm will be run. The reason that the line tracing ignores the other boids is that collision avoidance with them is already handled by the separation flocking rule.

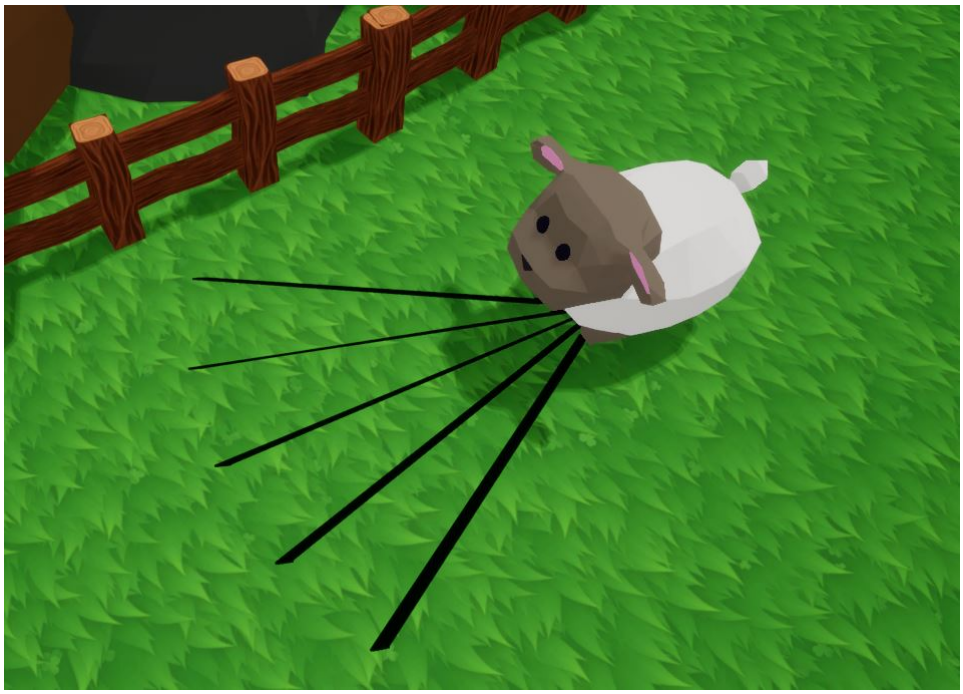


Figure 3.6: A sheep using linetracing to detect objects in its environment

When line tracing detects a potential collision, the boids will try to avoid collision with the object that the line trace hit. This is done by taking all the forces which are acting on the boid and removing the component of the acceleration that is parallel to the normal vector of the object on which a future collision is detected. This is often enough for the boid to move in another direction, but when a boid has a large flock behind it, its desire to follow the flock into the wall was still too high and further measures had to be taken. This was fixed by removing the same component as earlier but from the velocity vector instead. This made it so, at some point, the boid is unable to move into the wall even if it wanted to. These new vectors are calculated by taking the sum of all forces on the boid, as well as its velocity, and removing the component that is parallel to the normal vector from the object detected by the collision detection algorithm. The formula for removing the normal vector is the following:

$$vt = v - m(v \cdot m)$$

Where vt is the final force of the boid, v is the starting sum of forces, m is the normal vector of the object and $\cdot$ denotes the dot product (scalar product) operation. This function is visualised in Figure 3.7.

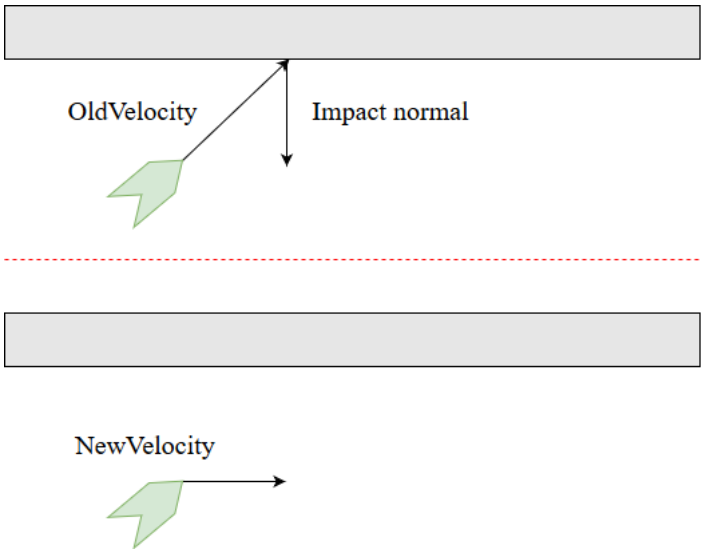


Figure 3.7: Velocity vectors before and after the collision avoidance algorithm is executed

3.6 Testing

The game was continuously tested by the members of the project group as part of the version control submission process. Additional testing in the form of simple scripted and unscripted user test has been performed ad hoc with users represented by peers at the university, family members, and acquaintances with an interest in games. The user tests allowed other people outside of the project to try the game and provide their feedback on it.

4

Results

4.1 Gameplay

When starting the game, the player's first view is a start screen containing a main menu with three options: play, controls and quit. This can be seen below in figure 4.1.

Figure 4.1: The game's start screen and the cursor icon.

When the player has pressed *play*, an overview of level one appears. This is the first time the player sees the actual world and the game interface, as seen in figure 4.2. The game interface contains the current score, the dog's stamina bar, the timer, the barricade controls, and the number of sheep needed in order to be able to advance to the next level.

Figure 4.2: The view when the player has pressed *play*.

In the top left, the user interface shows the number of points the player has scored as well as the number of sheep that have been herded into the next pasture. The top right shows instructions for how to place the barricades. In the bottom there is a meter for the dog sprint that goes down as you sprint and replenishes while the dog is walking or standing still. In the top there is a timer that shows the player how much time they have left before they will have had to move on to the next pasture. By pressing the Escape-key you can access the menu where you can restart the game.

In the world, the player can see a number of sheep that has spawned, the dog which the player controls, the fences surrounding level one, and some of the inaccessible areas as described in section 2.2.1. This can be seen in figure 4.3. Now, the player's task is to herd as many sheep as possible through the open gate, which leads to the next level. Before the player has herded enough sheep to the next level, only the sheep are able to enter through the gate. If the player tries to enter, the dog will be hindered by an *invisible wall*. Once the player has herded enough sheep, they can decide to go to the next level. If the time runs out before that, the end condition as described in section 2.2.4 is triggered, and thus the game is over.

Figure 4.3: One sheep is herded over to level two.

When the player decides to move over to level two, the camera switches to a camera for the new level as shown in figure 4.4 and the player is awarded additional time to complete the task. After the player has passed the gate, an animated gate closes off the area behind, making it impossible to go back.

Figure 4.4: Overview of level two.

Upon entering the new level, the herded sheep will join up with the flock already present in the new pasture. With the increased number of sheep in the level, the number of sheep needed for the next level in order to advance is raised as well. This happens for every new level that the player enters. With a higher level, the amount of points earned by herding sheep is increased.

4.2 Sheep flocking algorithm

The final flocking algorithm of this project consists of five different aspects, or rules, to make a realistic flocking simulation. Three originate from Reynolds' three rules, we refer to them as cohesion, separation and alignment described in further detail in section 2.3. The fourth rule allows the sheep to run away from the dog, which is highly influenced by another group's work explained in section 2.3.2 [8]. The last rule allows the sheep to realistically avoid obstacles. This algorithm is run on every sheep every game tick.

The final algorithm can be seen in listing 4.1, it works by first applying formula 2.2 which is line 1-9 in the listing, then on line 11-12 the velocity is updated and limited to an amplified max speed depending on the distance to the dog, as well as setting the velocity to zero if it is too low or directed behind the sheep. This first part is very much based on Reynolds algorithm and the modifications made in Barksten and Rydbergs thesis[8] as described in section 2.3.2. One difference from Barksten and Rydbergs algorithm is that cohesion follows the Reynolds way of only being applied on the nearby sheep instead of all sheep. Then the final rule of obstacle avoidance is applied using the updated velocity as can be seen on line 14 in listing 4.1. It is then added to the results of the first four rules and the velocity is limited in the same way as before on line 15.

Listing 4.1: Final flocking algorithm used in the game, sheep being the current sheep, the parameters are explained in table 3.1

```
1  combinedForces = ZeroVector
2  alignment = Align(sheep, nearbySheep)
3  combinedForces += ma * (1 + Sigmoid(distToDog, escapeRadius) * map)
       * alignment
4  cohesion = Cohesion(sheep, nearbySheep)
5  combinedForces += mc * (1 + Sigmoid(distToDog, escapeRadius) * mcp)
       * cohesion
6  separation = Separation(sheep, nearbySheep)
7  combinedForces += ms * (1 + Sigmoid(distToDog, escapeRadius) * msp)
       * separation
8  escape = Escape(sheep, dog)
9  combinedForces += me * escape
10 ampedMaxSpeed = maxSpeed * (1 + Sigmoid(distToDog, escapeRadius) *
       1.5)
11 velocity = LimitVelocity(sheep, combinedForces, ampedMaxSpeed)
12 avoid = AvoidObstacles(sheep)
13 velocity = LimitVelocity(sheep, velocity + mAvoid * avoid,
       ampedMaxSpeed)
```

4.3 Performance

In the first iteration of creating the flocking simulation, the calculations for each individual boid was done on a single thread which is an inefficient way of simulating a flock, as the whole workload would be put onto one thread. This was necessary to do in the beginning in order to confirm that the flocking algorithm worked properly without having to worry about multithreading issues.

Stress testing was performed to see how many sheep the game could handle before the frame rate dropped under 30FPS. The number of sheep that could be simulated in the single threaded implementation at that time before adding much in terms of graphics was around 400 and when applying multithreading around 800 sheep in the development environment was possible. However, the improvement is throttled because using draw calls in external threads is not supported in UE4. Therefore, all the draw calls have to be done sequentially in the main game thread.

Table 4.1: Table of average values from profile testing on the development build

Amount of sheep	**avg GPU**	**avg RT**	**avg GT**	**avg Mesh draw calls**	**avg FPS**
100	16.87ms	11.86ms	11.06ms	2909.37	59.28
200	20.15ms	15.63ms	18.62ms	5102.03	49.62
300	30.36ms	27.24ms	27.76ms	7561.2	32.92
400	41.22ms	33.77ms	39.73ms	11944	24.27
500	54.56 ms	43.15 ms	51.83 ms	12392.78	18.32
600	64.44ms	49.74ms	62.99ms	14061.45	15.51

Table 4.1 displays various performance numbers from UE4 achieved in the final game when running the game in development mode on a Windows 10 machine running on an i5-6400 CPU @ 2.70GHz and a NVIDIA GeForce GTX 960. *Avg GPU* is the average time per frame on the GPU thread, *Avg RT* is the average time per frame on the CPU render thread, *Avg GT* is the average time per frame on the CPU game thread [27]. These are threads that are built into UE4 and where most of the game calculations take place. When optimising for performance, the largest of these three should be targeted since it will hold back the total performance and determine the FPS. *Avg mesh* draw calls are the amount of mesh draw calls per frame, this number is very high in this game even with low amount of sheep and there is probably space for a lot of optimising here. When further breaking down the GPU thread profiling it could be seen that a lot of time is spent on lightning and post processing which could be optimised but was not prioritised.

Table 4.2: Table listing the amount of sheep used in a game session on a shipping build, followed by the amount of frames, minimum, maximum and average FPS

Amount of sheep	Frames	FPS Min	FPS Max	FPS Avg
300	3660	54	63	61
400	3123	49	56	52.05
500	2515	39	44	41.917
600	2100	31	37	35
700	1773	27	32	29.55
800	1541	23	28	25.683
900	1382	21	25	23.033
1000	1135	18	23	21.917
1100	1162	16	21	19.367
1200	1058	16	19	17.633

In order to know how well the game will perform outside of the development build of the game, performance tests were done on the shipping build of the game. It is a 64-bit build of the game which will be the final build that will be released for the average consumer. These values are important to test in order to see the performance of the game on the end users systems. Table 4.2 displays the frame rate results when running the game on this build. The average frames per second are improved as the amount of sheep increases compared to the development build as seen in table 4.1, this is because the shipping build does not have any debugging tools running in the background.These tests were done on an external program named FRAPS which is a screen capture, recording and benchmarking software.

The performance goal of this project was to be able to have 1000 sheep in-game. The average FPS when spawning 1000 sheep is 21.917 in the shipping-configuration build as seen in table 4.2. The amount of FPS for a playable experience is usually 30 FPS, therefore the player might have a slightly unpleasant experience with 1000 sheep. It is worth mentioning that the levels were not enjoyable when having 1000 sheep as they covered most of the surface area of the entire level.

5
Discussion

Some things that were initially considered for the scope had to be abandoned due to a lack of time and a higher than expected complexity of some tasks. Some features, such as multiplayer capability and in-game power-ups, were too inessential to have time to be implemented. Other features, such as parallelisation and spatial data structures, were implemented but could not reach their full potential due to both a lack of time and technical limitations with Unreal Engine 4. During the course of this project many decisions amounted to one specific dilemma: Should the focus be on the gameplay aspect or on the flocking optimisation aspect? While the subject is interesting no matter which focus area you choose, it soon became clear that the entire group would rather have a polished and fun game than a highly optimised flocking algorithm. In the end, the result was a satisfying and fully playable game demo.

5.1 Flocking Implementation

The three rules for flocking work well in this implementation, just as Craig Reynolds showed it would in his 1987 article and many after him, but it requires you to painstakingly balance and test different parameters by hand until you reach a satisfying result. There has been work exploring the possibilities of optimising these parameters with an algorithm instead of by hand, either by using a Genetic Algorithm or Particle Swarm Optimisation [28]. They both work by setting up cost functions judging the behaviour of the flock and trying to minimise the total cost by changing the parameters. By using any of these methods the difficulty of balancing the parameters manually is removed but instead the difficulty is to create good cost functions. Using parameter optimisation would likely have resulted in better flocking behaviour and might even have saved time if only such a thing had been implemented earlier and used from the beginning. However, by the time the group identified this problem, too much time had already been invested into manually tuning the parameters and that idea was scrapped.

When tuning the parameters it quickly became obvious that a well functioning set of parameters one flock size, would not necessarily work for flocks of other sizes. In other words, a larger flock tended to clump together and needed a higher separation while the same separation would cause a smaller flock to be too spread out. So while our implementation worked well, there are many ways to improve the flocking itself. For example, creating a more realistic visual field that is more like that of

a real sheep, i.e. they do not take sheep directly behind them into consideration. Adding separate rules for specific tasks in this game, such as making the sheep run through the gate more realistically, might have improved the flocking algorithm as well as the general feel of the game. This specific example could be achieved by implementing and placing an attractor in the adjacent pasture.

The first rendition of collision avoidance was done by adding two vectors in front of each boid which could detect collision with walls and obstacles. When a collision was detected, the distance to the obstacle was used to calculate which direction an acceleration should be applied in order to achieve realistic avoidance behaviour. This implementation resulted in an inconsistent behaviour however and to improve it we ended up increasing the number of vectors to the point where it impacted the game's performance. It was obvious that this implementation needed a lot of rework and after consulting our supervisor, a better solution was worked out that essentially did the opposite of the previous implementation. Instead of adding an acceleration away from the obstacle, the new solution opted for removing the acceleration and velocity directed towards the obstacle. This version required only one vector and gave promising results despite needing changes to the map and other parts of the code.

5.2 Gameplay

The relationship between camera distance and flock size in combination with the design decision of an overview static camera turned out to introduce an unexpected problem. Introducing as many sheep as possible is desired both in order to utilise the optimisation efforts and because the flocking behaviour is more apparent with a larger flock. More sheep means that each pasture has to be larger in order to fit them all, a larger pasture in turn means that the camera has to be further away in order to fit it all into the frame. But as the camera moves further away and everything becomes smaller the game becomes increasingly hard and more frustrating to play. This could of course be solved with a closer positioned camera that actively follows the dog but this would likely worsen the flocking effect as well as introduce other gameplay issues such as not easily being able to see which way to go.

As the game is now, there is not much that motivates the player to get a high score instead of just get to an as high level as possible. To make it more motivating for the player to get high scores, an option discussed is the possibility of awarding assets or abilities to the player. Assets or abilities suitable in this game could be more stamina for the dog or more time per level, which in turns would have made it simpler to advance to the next level faster. To regulate the faster level advancing, the default time that the player gets could be decreased, the dog's and sheep's speeds could be decreased or the number of sheep that need to be herded to the next level could be increased.

As mentioned in section 2.2, the impression of a well-balanced game is important in order to make the game entertaining to play. The game elements in this game that

can be balanced against each other is the time, the dog's stamina, the barricades that can be placed in the world, the amount of sheep needed to be able to advance to the next level and how fast the sheep and the dog is. The balancing of these elements should have been tested through more formal user tests than occasional tests on group members and peers in order to get the perfect balance. More time could have been spent balancing the game, but unfortunately the focus had to be to ensure a stable game experience without crashes.

5.3 Parallelisation

The parallelisation of the simulation was going to be one of the main methods of optimising the flocking algorithm. However, due to limitations in Unreal Engine 4 it is not possible to perform draw calls in external threads. A proposed solution for this was to merge all sheep meshes into one mesh in order to perform only one draw call for all the sheep. However due to the fact that each sheep is derived from an actor class, we cannot perform this operation. Merging actors is an operation that can be done in Unreal Engine 4 but it can only be done inside the editor when constructing maps for the game and is not something that can be done repeatedly in C++ code.

Unlike Unity, which is an alternative game engine and competitor to Unreal Engine 4, Unreal Engine 4 has no draw call batching for actors. This means that there is no easy way of drawing many actors simultaneously [29]. By being able to draw many actors simultaneously, the GPU can be used more efficiently to achieve greater performance. In order to achieve similar GPU efficiency in UE4 Instanced Static Meshes can be used instead. This means that for the entire flock there can only be one actor instead of an actor for each and every sheep.

Since most of the game has been built without the group knowing of this limitation, much of the code would have had to be refactored in order to achieve the greater performance of instanced meshes. Instanced static meshes are however static which means that they cannot be vertex animated, they can only be scaled, translated and rotated in real time. By using actors for every sheep like in the current implementation, skeletal meshes can be used which have detailed vertex animations which allows for better presentation for the game.

5.4 Uniform Grid

The current implementation for getting nearby cells is not the most optimal. When a sheep is on the edge of a cell and is heading to a nearby cell, taking into account all of the surrounding cells is unnecessary as the cells that are behind the sheep are most likely not going to affect the flocking forces and collision detection of the sheep. The sampling pool can therefore possibly be reduced even more by only looking for the nearby cells which the sheep's velocity vector is pointing towards, without possibly affecting the flocking and collision algorithms.

5.5 Ethical aspects

With any engineering project one has to consider possible ethical implications. The outcome of this project should not have any direct ethical implications and further investigation into ethics was not included in the scope past this discussion section. The wider research area of flocking behaviours could be applied in a vast range of areas, spanning from graphics and games to drones designed to be deployed in war zones. The following sections introduce the application of flocking behaviours in a couple of areas which may have ethical implications.

5.5.1 Military units

Unmanned combat aerial vehicles known as UCAVs are becoming increasingly common in modern warfare, these drones are operated without a human pilot on board and commonly carries missiles which are fired by a remote operator in so-called drone strikes. The level of autonomy in these drones vary, but the area of artificial intelligence as a military tool is under constant development and flocking behaviours are being researched by both the Russian and the US army as a mean to increase the efficiency of ground units and unmanned aircraft infantry in urban environments [30], [31].

While unmanned military units may have a positive impact on soldiers who can be kept out of the field, the group considers this application of flocking behaviours to be problematic from an ethical perspective. The U.S emphasise the precision of drone strikes and claim that extremely few civilians have been killed in the strikes, but there is a large degree of uncertainty surrounding these claims as the U.S government resists openly disclosing official drone strike data [32].

5.5.2 Robotic bees

With natural pollinators decreasing in numbers researchers are exploring the possibilities of artificial pollinators, such as RoboBees. These airborne robots in the size of bees are being developed by a research robotics team at Harvard University [33]. The ultimate goal of the RoboBees project is to develop colonies of autonomous and wireless miniature robots, however, the microchips of today are too big to allow the project to reach this goal (as of 2013). Once this hardware problem has been overcome it is believed that large groups of RoboBees could be used as artificial pollinators by utilising swarming intelligence.

The research field of artificial pollinators is fascinating and if pollinators could be produced under economically and environmentally sound conditions it is tempting to think that they could be of great benefit [34]. The question becomes more complex when considering the pollinators long term effects on the biosphere. There is some evidence that artificial pollinators could be detrimental to the ecosystem they act within by disrupting local species network interactions and even if they would work,

you have to question if we should not be focusing on preservation of biodiversity rather than replace it.

5.6 Further Development

If the group would have had more time, some changes and extensions of the game design would have been made. With the current implementation, herding a lot of sheep into the next level at the same time is difficult. As shepherds in real life make use of several dogs to get the flock of sheep where they want, turning this game into a multiplayer game where every player have control over their own dog would have been fun to try out.

Other things that were discussed regarding the gameplay at the beginning of the project were having power-ups, objects that instantly benefit or add extra abilities to the game character as a game mechanic, available for the player to pick up. These power-ups could make the dog regain its stamina faster, award extra points or more time. The implementation of attractors, invisible entities that the sheep want to walk to, could have helped with getting sheep into areas that were especially difficult, such as gates.

As discussed in section 5.3 and 4.3, the parallelisation of this game did not go as well as the group had hoped for. While using static meshes to represent the sheep would have allowed for better optimisation with multithreading, static meshes do not allow skeletal animations. A lot of problems with the optimisation could be boiled down to issues and limitations with Unreal Engine 4 as the game engine. While it, on one hand, makes graphics easy to handle and can render a beautiful game with very little work it lacked some features that would have made other areas easier to handle, such as spatial data structures and parallelisation, that other game engines include. Unreal Engine 4 is not a bad game engine but had the group known about its limitation earlier in the project, Unity would have been considered a stronger contender.

Part of the purpose was to develop an entertaining game, whether a game is fun or not is very subjective. There should have been more user testing in order to be able to discuss the fulfilment of that part of the purpose more and come to a solid conclusion. A few user tests were performed during an exhibition of the project as well as on peers of the group during development. While most of these testers found the game fun and polished, the tests are not comprehensive or unbiased enough in order to be used for a conclusion. More user testing would also have been helpful for balancing parameters and implementing game features based on more input than just the groups' opinions and research.

6
Conclusion

Considering the purpose of this project was to "develop an entertaining procedural game with flocking behaviour as the central feature" it can be difficult to evaluate the success of the result. As mentioned in the discussion an entertaining game is subjective and hard to determine, therefore we can not come to a definitive conclusion about whether the game is fun or not. Building a game around flocking behaviour as the central game mechanic was found to work well. The procedural nature of the flocking behaviour results in a reactive AI, making for a unique player experience every time. As mentioned in sections 1.5 and 5, the size of the flock had a lower priority while the playability and realism of the behaviour had a higher focus. A few dead-ends were encountered in the implementation of parallelisation and spatial data-structures, leading to unexpected results. On the other hand, Unreal Engine 4 allowed for implementing a lot of smart functions into the game using the built-in methods of the engine.

When going through the process of creating a simple but functioning game the group have learned a lot about both flocking behaviour and game design. To continue on this work or create something similar, it may be beneficial to review what has been done in this project. Some possible extensions to this implementation of the game may be a multiplayer version where the players would play as one herding dog each and teaming up to accomplish the goals of the game. Since herding in real life usually is done by multiple dogs, this could make the game more enjoyable and realistic. An additional feature could be power-ups that would give your dog new abilities or boost his existing ones. These power-ups could be acquired either temporarily by collecting them on the map or permanently by buying them between the runs in a shop.

The members of the group had little to no experience working with either the programming language or the game engine, as a result this project has been a great learning opportunity and *Baa!* turned out to be a satisfying game.

7

References

[1] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, aug 1987. doi: 10.1145/37402.37406, [Online]. Available: https://dl.acm.org/citation.cfm?id=37406, Accessed on: March 05, 2019.

[2] J. M. E. Gabbai, "Complexity and the aerospace industry: Understanding emergence by relating structure to performance using multi- agent systems," Engineering Doctorate, Faculty of Engineering and Physical Sciences, School of Electrical and Electronic Engineering, Manchester, England, 2005. pp. 66, doi: 10.1.1.141.2137, [Online]. Available: http://gabbai.com/academic/complexity-and-the-aerospace-industry-understanding-emergence-by-relating-structure-to-performance-using-multi-agent-systems, Accessed on: 25 April, 2019.

[3] J. Hagelbäck, "Hybrid pathfinding in starcraft," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, pp. 319–324, Dec 2016. doi: 10.1109/TCIAIG.2015.2414447, [Online]. Available: https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=4804728, Accessed on: 20 April, 2019.

[4] H. P. Zhang, A. Be'er, E.-L. Florin, and H. L. Swinney, "Collective motion and density fluctuations in bacterial colonies," *Proceedings of the National Academy of Sciences*, vol. 107, no. 31, pp. 13626–13630, 2010. doi: 10.1073/pnas.1001651107, [Online]. Available: https://www.pnas.org/content/107/31/13626, Accessed on: 22 April 2019.

[5] Unity, "Game engines—how do they work?." [Online]. Available: https://unity3d.com/what-is-a-game-engine, 2019. Accessed: 18 April, 2019.

[6] J. Schell, *The art of game design: a book of lenses*. Elsevier, 2010.

[7] B. Staffan and J. Holopainen, *Patterns in game design*. Charles River Media, 2006. Boston, United States.

[8] M. Barksten and D. Rydberg, "Extending reynolds' flocking model to a simulation of sheep in the presence of a predator," 2013.

[9] P. Barbosa and I. Castellanos, *Ecology of Predator-Prey Interactions*, p. 109. Oxford University Press, 2014.

[10] Agriculture Knowledge Centre, "Understanding sheep behaviour." [Online]. Available: https://www.sksheep.com/documents/Ex_Understanding_Sheep_Behaviour.pdf, 2007. Accessed: 16 April, 2019.

[11] A. J. King, A. M. Wilson, S. D. Wilshin, J. Lowe, H. Haddadi, S. Hailes, and A. J. Morton, "Selfish-herd behaviour of sheep under threat," *Current Biology*, vol. 22, no. 14, pp. R561 – R562, 2012. doi: https://doi.org/10.1016/j.cub.2012.05.008, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0960982212005295, Accessed on: 04 April 2019.

[12] M. . Pillot, J. Gautrais, P. Arrufat, I. D. Couzin, R. Bon, and J. . Deneubourg, "Scalable rules for coherent group motion in a gregarious vertebrate," *PLoS ONE*, vol. 6, no. 1, 2011. doi: https://doi.org/10.1371/journal.pone.0014487, [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0014487, Accessed on: 10 April, 2019.

[13] Mark Joselli et al., "A flocking boids simulation and optimization structure for mobilemulticore architectures." [Online]. Available: https://pdfs.semanticscholar.org/4183/faeef708a56b988742b5572fce9174caec7b.pdf, 2012. Accessed: 15 February, 2019.

[14] S. Green, "Particle simulation using cuda." [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/particles.pdf, 2010. Accessed: 15 February, 2019.

[15] R. Hidayat, D. Spataro, E. D. Giorgio, W. Spataro, and D. DAmbrosio, "Multi-agent system with multiple group modelling for bird flocking on gpu," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 680–685, Feb 2016. doi: 10.1109/PDP.2016.112, [Online]. Available: https://ieeexplore.ieee.org/document/7445408, Accessed on: April 03, 2019.

[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008. doi: 10.1109/JPROC.2008.917757, [Online]. Available: https://ieeexplore-ieee-org.proxy.lib.chalmers.se/document/4490127/citations#citations, Accessed on: April 04, 2019.

[17] Epic Games, "We succeed when you succeed." [Online]. Available: https://www.unrealengine.com/en-US/what-is-unreal-engine-4, 2019. Accessed: 15 April, 2019.

[18] M. Noland, "Unreal property system (reflection)." [Online]. Available: https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection, 2014. Accessed: 15 April, 2019.

[19] F. Silva, "Lowpoly farm animals." [Online]. Available: https://assetstore.unity.com/packages/3d/characters/animals/lowpoly-farm-animals-81478. Accessed: 15 May, 2019.

[20] Kev92, "Low poly forest pack." [Online]. Available: https://www.blendswap.com/blends/view/76557. Accessed: 15 May, 2019.

[21] Jaks, "Low poly forest pack." [Online]. Available: https://jaks.itch.io/lowpolyforestpack. Accessed: 15 May, 2019.

[22] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018.

[23] Epic Games, "Ambient occlusion." [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/AmbientOcclusion. Accessed: 15 May, 2019.

[24] Epic Games, "Actors and geometry." [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Actors, 2019. Accessed: 15 April, 2019.

[25] Epic Games, "Pawn." [Online]. Available: https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn, 2019. Accessed: 15 April, 2019.

[26] Epic Games, "Sphereoverlapactors." [Online]. Available: http://api.unrealengine.com/INT/BlueprintAPI/Collision/SphereOverlapActors/, 2018. Accessed: 15 May, 2019.

[27] Epic Games, "Performance and profiling overview." [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Performance/Overview. Accessed: 15 May, 2019.

[28] S. Alaliyat, H. Yndestad, and F. Sanfilippo, "Optimisation of boids swarm model based on genetic algorithm and particle swarm optimisation algorithm (comparative study)," *Proceedings - 28th European Conference on Modelling and Simulation, ECMS 2014*, 05 2014. doi: 10.7148/2014-0643, [Online]. Available: https://www.researchgate.net/publication/268077894-Optimisation-Of-Boids-Swarm-Model-Based_On-Genetic-Algorithm-And-Particle-Swarm-Optimisation-Algorithm-Comparative-Study, Accessed on: 20 April, 2019.

[29] Unity, "Unity draw call batching." [Online]. Available: https://docs.unity3d.com/Manual/DrawCallBatching.html, 2019. Accessed: 16 April, 2019.

[30] Congressional Research Service, "Artificial intelligence and national security." [Online]. Available: https://fas.org/sgp/crs/natsec/R45178.pdf, 2019. Accessed: 10 February, 2019.

[31] Defense Advanced Research Projects Agency, "Offset envisions swarm capabilities for small urban ground units." [Online]. Available: https://www.darpa.mil/news-events/2016-12-07, 2016. Accessed: 10 February, 2019.

[32] Human Rights Clinic at Columbia Law School , "Counting drone strike deaths," 10 2012. [Online]. Available: https://www.law.columbia.edu/sites/default/files/microsites/human-rights-institute/files/COLUMBIACountingDronesFinal.pdf.

[33] R. Wood, R. Nagpal, and G.-Y. Wei, "Flight of the robobees." [Online]. Available: https://ssr.seas.harvard.edu/publications/flight-robobees, 2013. Accessed: 10 February, 2019.

[34] S. G. Potts, P. Neumann, B. Vaissière, and N. J. Vereecken, "Robotic bees for crop pollination: Why drones cannot replace biodiversity," *Science of The Total Environment*, vol. 642, pp. 665 – 667, 2018. doi: https://doi.org/10.1016/j.scitotenv.2018.06.114, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0048969718321909, Accessed on: 25 April 2019.